C Language Portability

A. T. Short

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft.

CONTENTS

LIST OF FIGURES

C Language Portability

1. Introduction

The C language is defined in the appendix to The C Programming Language [1]. This definition leaves many details to be decided by individual implementations of the language. It is those incompletely specified features of the language that detract from its portability and that should be studied when attempting to write portable C code.

Most of the issues affecting C portability arise from differences in either target machine hardware or compilers. C was designed to compile to efficient code for the target machine (initially a PDP-11) and so many of the language features not precisely defined are those that reflect a particular machine's hardware characteristics.

This document highlights the various aspects of C that may not be portable across different machines and compilers. It also briefly discusses the portability of a C program in terms of its environment, which is determined by the system calls and library routines it uses during execution, file pathnames it requires, and other items not guaranteed to be constant across different systems.

The C language has been implemented on many different computers with widely different hardware characteristics, varying from small 8-bit microprocessors to large mainframes. This document is largely concerned with the portability of C code in the XENIX programming environment. This is a more restricted problem to consider since all XENIX systems to date run on hardware with the following basic characteristics:

- ASCII character set
- 8-bit bytes
- 2-byte or 4-byte integers
- Two's complement arithmetic

None of these features is required by the formal definition of the language, nor is it true of all implementations of C. However, the remainder of this document is largely devoted to those systems where these basic assumptions hold.

The C language definition contains no specification of how input and output is performed. This is left to system calls and library routines on individual systems. Within XENIX systems there are a large number of system calls and library routines which can be considered portable. These are

described briefly in a later section.

This document is not intended as a C language primer, for which [1] should be used. It is assumed here that the reader is familiar with C, and with the basic architecture of common microprocessors.

2. Source Code Portability

We are concerned here with source code portability, which means that programs can be compiled and run successfully on different machines without alteration.

Programs can be written to achieve this goal using several techniques. The first is to avoid using inherently nonportable language features. Secondly, any nonportable interactions with the environment, such as I/O to nonstandard devices should be isolated, and possibly passed as an argument to the program at run time. For example programs should not, in general, contain hard-coded file pathnames except where these are commonly understood to be present in all XENIX systems. (an example might be /etc/passwd).

Files required at compile time (i.e., include files) may also introduce nonportability if the pathnames are not the same on all machines. However in some cases the use of include files to contain machine parameters can be used to make the source code itself portable.

2.1 Machine Hardware

As mentioned above, most nonportable features of the C language are due either to hardware differences in the target machine or to compiler differences. This section lists the more common hardware differences encountered on XENIX systems and some language features to beware of.

2.1.1 Byte Length

The length of the char data type is not defined in the language, other than that it must be sufficient to hold all members of the machine's character set as positive numbers. Within the scope of this document we will consider only 8-bit bytes, since this is the byte size on all XENIX systems.

2.1.2 Word Length

The definition of C makes no mention of the size of the basic data types for a given implementation. These generally follow the most natural size for the underlying machine. It is safe to assume that short is no longer than long. Beyond that no assumptions are portable. For example on the PDP-11 short is the same length as int, whereas on the VAX long is the same length as int.

Programs that need to know the size of a particular data type should avoid hard-coded constants where possible. Such information can usually be written in a fairly portable way. For example the maximum positive integer (on a two's complement machine) can be obtained with:

```
#define MAXPOS  ((int)(((unsigned)~0) >> 1))
```

This is usually preferable to something like:

```
#ifdef PDP11
#define MAXPOS 32767
#else
        ...
#endif
```

Likewise to find the number of bytes in an int use sizeof(int) rather than 2, 4, or some other nonportable constant.

2.1.3 Storage Alignment

The C language defines no particular layout for storage of data items relative to each other, or for storage of elements of structures or unions within the structure or union.

Some CPU's, such as the PDP-11 and M68000 require that data types longer than one byte be aligned on even byte address boundaries. Others, such as the 8086 and VAX-11 have no such hardware restriction. However, even with these machines, most compilers generate code that aligns words, structures, arrays and long words, on even addresses, or even long word

addresses. On the VAX-11, the following code sequence gives "8", even though the VAX hardware can access an int (a 4-byte word) on any physical starting address:

```
struct s_tag {
        char c;
        int  i;
};
printf("%d\n",sizeof(struct s_tag));
```

The principal implications of this variation in data storage are twofold:

- Data accessed as nonprimitive data types is not portable,
- Code that makes use of knowledge of the layout on a particular machine is also not portable.

Thus unions containing structures are nonportable if the union is used to access the same data in different ways. Unions are only likely to be portable if they are used simply to have different data in the same space at different times. For example, if the following union were used to obtain four bytes from a long word, there's no chance of the code being portable:

```
union {
        char c[4];
        long lw;
} u;
```

The sizeof operator should always be used when reading and writing structures:

```
struct s_tag st;

   ...

write(fd, &st, sizeof(st));
```

This ensures portability of the source code. It does NOT produce a portable data file. Portability of data is discussed in a later section.

Note that the sizeof operator returns the number of bytes an object would occupy in an array. Thus on machines where structures are always aligned to begin on a word boundary in memory, the sizeof operator will include any necessary padding for this in the return value, even if the padding occurs after all useful data in the structure. This occurs

whether or not the argument is actually an array element.

2.1.4 Byte Order in a Word

The variation in byte order in a word between machines affects the portability of data between machines more than the portability of source code. However any program that makes use of knowledge of the internal byte order in a word is not portable. For example, on some PDP-11 systems there is an include file misc.h which contains the following structure declaration:

```
/*
* structure to access an
* integer in bytes
*/
struct {
        char    lobyte;
        char    hibyte;
};
```

With certain less restrictive compilers this could be used to access the high and low order bytes of an integer separately, and in a completely nonportable way. The correct way to do this is to use mask and shift operations to extract the required byte:

```
#define LOBYTE(i) (i & 0xff)
#define HIBYTE(i) ((i >> 8) & 0xff)
```

Note that even this is only applicable to machines with two bytes in an int.

One result of the byte ordering problem is that the following code sequence will not always perform as intended:

```
int c = 0;

read(fd, &c, 1);
```

On machines where the low order byte is stored first, such as the PDP-11, the value of c will be the byte value read. On other machines the byte is read into some byte other than the low order one, and the value of c is different.

In all cases the compiler will generate coercions such that mixed char and int expressions will work correctly. Thus it is reasonable to write the following:

```
int i;
 ..
 ..
putchar(i);
```

This is safe because it avoids taking the address of the data item containing the character.

2.1.5 Bitfields

Bitfields are not implemented in all C compilers. When they are, a number of restrictions apply:

- No field may be larger than an int.
- No field will overlap an int boundary. If necessary the compiler will leave gaps and move to the next int boundary.

The C language makes no guarantees about whether fields are assigned left to right, or right to left in an int. Thus while bitfields may be useful for storing flags, and other small data items, their use in unions to disect bits from other data is definitely nonportable.

To ensure portability no individual field should exceed 16 bits.

2.1.6 Pointers

The C language is fairly generous in allowing manipulation of pointers, to the extent that most compilers will not object to nonportable pointer operations. The lint program is particularly useful for detecting questionable pointer assignments and comparisons.

The common nonportable use of pointers is where a pointer to one data type is cast to be a pointer to a different data type. This almost always makes some assumption about the internal byte ordering and layout of the data type, and is therefore nonportable. For example, in the following code the ordering of the bytes from the long in the byte array is not portable:

```
char c[4];
long *lp;

lp = (long *)&c[0];
*lp = 0x12345678L;
```

The lint program will issue warning messages about such uses of pointers. Very occasionally it is necessary and valid to write code like this. An example is when the malloc() library routine is used to allocate memory for something other than type char. The routine is declared as type

```
char *
```

and so the return value has to be cast to the type to be stored in the allocated memory. If this type is not char * then lint will issue a warning concerning illegal type conversion. In addition, the malloc() routine is written to always return a starting address suitable for storing all types of data, but lint does not know this, so it gives a warning about possible data alignment problems too. In the following example, malloc() is used to obtain memory for an array of 50 integers. The code will attract a warning message from lint. There is nothing which can be done about this.

```
extern char *malloc();
int *ip;

ip = (int *)malloc(50);
```

2.1.7 Address Space

The address space available to a program running under XENIX varies considerably from system to system. On a small PDP-11 there may be only 64k bytes available for program and data combined (although this can be increased - see 23fix(1)). Larger PDP-11's, and some 16 bit microprocessors allow 64k bytes of data, and 64k bytes of program text. Other machines may allow considerably more text, and possibly more data as

well.

Large programs, or programs that require large data areas may have portability problems on small machines. The table below indicates the code and data sizes available on some XENIX systems:

XENIX System Address Space Limits

	PDP-11* 23,40,60	PDP-11 44,45,70	Z8000	8086	M68000	VAX-11
Code	48k	64k	64k	110k	4meg	16meg
Data	48k	56k	60k	60k	4meg	16meg

* Note PDP-11 23,40,60 Code + Data < 56k.

XENIX 3.0 has a ulimit(2) system call which returns the maximum allowed argument to the brk(2) system call. This may be useful to avoid hard-coding maximum runtime memory allocations.

2.1.8 Character Set

We have said that we are concerned here mainly with the ascii character set. The C language does not require this however. The only requirements are:

- All characters fit in the char data type.
- All characters have positive values.

In the ascii character set, all characters have values between zero and 127. Thus they can all be represented in 7 bits, and on an 8 bits per byte machine are all positive regardless of whether char is treated as signed or unsigned.

There is a set of macros defined under XENIX in the header file /usr/include/ctype.h which should be used for most tests on character quantities. Not only do they provide some insulation from the internal structure of the character set, their names are more meaningful than the equivalent line of code in most cases. Compare

```
if(isupper(c))
```

to

```
if((c >= 'A') && (c <= 'Z'))
```

With some of the other macros, such as isxdigit() to test for a hex digit, the advantage is even greater. Also, the internal implementation of the macros makes them more efficient than an explicit test with an 'if' statement.

2.2 Compiler Differences

There are a number of C compilers running under XENIX. On PDP-11 systems there is the so called "Ritchie" compiler [2]. Also on the 11, and on most other systems, there is the Portable C Compiler [3]. The various compilers give rise to a number of nonportable features, described in the following sections.

2.2.1 Signed/Unsigned char, Sign Extension

The current state of the signed vs unsigned char problem is best described as unsatisfactory. The problem is completely explained and discussed in [4], so that material is not repeated here.

The sign extension problem is one of the more serious barriers to writing portable C, and the best solution at present is to write defensive code which does not rely on particular implementation features. [4] suggests some ways to do this.

2.2.2 Shift Operations

The left shift operator, <<, shifts its operand a number of bits left, filling vacated bits with zero. This is a so-called logical shift.

The right shift operator, >>, when applied to an unsigned quantity, performs a logical shift operation. When applied to a signed quantity, the vacated bits may be filled with zero (logical shift) or with sign bits (arithmetic shift). The decision is implementation dependent, and code which uses knowledge of a particular implementation is nonportable.

The PDP-11 compilers use arithmetic right shift. Thus to avoid sign extension it is necessary to either shift and mask out the appropriate number of high order bits, or to use a divide operator which will avoid the problem completely:

```
char c;

For  c >> 3;     use:     (c >> 3) & 0x1f;
                 or:      c / 8;
```

To maintain portability, always use '>>' to right shift an unsigned integer. Use divide to right shift a signed number. Where possible the C compiler will transform a signed divide by a power of two into an arithmetic right shift. It is not safe to assume that

```
x >> 1
```

divides by two. On many machines,

```
(-1) >> 1
```

will yield -1.

2.2.3 Identifier Length

The use of long identifier names will cause portability problems with some compilers. There are three different cases to be aware of:

- C Preprocessor Symbols
- C Local Symbols
- C External Symbols

The loader used may also place a restriction on the number of unique characters in C external symbols.

Symbols unique in the first six characters are unique to most C language processors.

2.2.4 Register Variables

The number and type of register variables in a function depends on the machine hardware and the compiler. Excess and invalid register declarations are treated as nonregister declarations, which should not cause a portability problem. On a PDP-11, up to three register declarations are significant, and they must be of type int, char, or pointer. ([1] Page 81). Whilst other machines/compilers may support declarations such as register unsigned short this should not be relied upon.

Since the compiler ignores excess register keywords, register type variables should always be declared in their

order of importance. register type. Then the ones the compiler ignores will be the least important.

2.2.5 Type Conversion

The C language has some rules for implicit type conversion ([1] Page 41). It also allows explicit type conversions by type casting. The most common portability problem arising from implicit type conversion is unexpected sign extension. This is a potential problem whenever something of type char is compared with an int.

For example

```
char c;

if(c == 0x80)
        ...
```

will never evaluate true on a machine that sign extends since c is sign extended before the comparison with 0x80, an int.

The only safe comparison between char type and an int is the following:

```
char c;

if(c == 'x')
        ...
```

This is reliable since C guarantees all characters to be positive. The use of hard-coded octal constants is subject to sign extension. For example the following program prints ff80 on a PDP-11:

```
main()
{
        printf("%x0,'\200');
}
```

Type conversion also takes place when arguments are passed to functions. Types char and short become int. Once again machines that sign extend char can give surprises. For example the following program gives -128 on the PDP-11.

```
char c = 128;
printf("%d\n",c);
```

This is because c is converted to int before passing on the stack to the function. The function itself has no knowledge

of the original type of the argument, and is expecting an int. The correct way to handle this is to code defensively and allow for the possibility of sign extension:

```
char c = 128;
printf("%d\n", c & 0xff);
```

2.2.6 Functions With Variable Number of Arguments

Functions with a variable number of arguments present a particular portability problem if the type of the arguments is variable too. In such cases the code is dependent upon the size of various data types.

In XENIX there is an include file, /usr/include/varargs.h, that contains macros for use in variable argument functions to access the arguments in a portable way:

```
typedef char *va_list;
#define va_dcl int va_alist;
#define va_start(list) list = (char *) &va_alist
#define va_end(list)
#define va_arg(list,mode) ((mode *)(list += sizeof(mode)))[-1]
```

The va_end() macro is not currently required. The use of the other macros will be demonstrated by an example of the fprintf() library routine. This has a first argument of type FILE *, and a second argument of type char *. Subsequent arguments are of unknown type and number at compilation time. They are determined at run time by the contents of the control string, argument 2.

The first few lines of fprintf() to declare the arguments and find the output file and control string address could be:

```
#include <varargs.h>
#include <stdio.h>

int
fprintf(va_alist)
va_dcl;
{
        va_list ap;     /* pointer to arg list   */
        char *format;
        FILE *fp;

        va_start(ap);   /* initialize arg pointer */
        fp = va_arg(ap, (FILE *));
        format = va_arg(ap, (char *));

                .....
}
```

Note that there is just one argument declared to fprintf(). This argument is declared by the va_dcl macro to be type int, although its actual type is unknown at compile time. The argument pointer, ap, is initialized by va_start() to the address of the first argument. Successive arguments can be picked from the stack so long as their type is known using the va_arg() macro. This has a type as its second argument, and this controls what data is removed from the stack, and how far the argument pointer, ap, is incremented. In fprintf(), once the control string is found, the type of subsequent arguments is known and they can be accessed sequentially by repeated calls to va_arg(). For example, arguments of type double, int *, and short, could be retrieved as follows:

```
double dint;
int *ip;
short s;

dint = va_arg(ap, double);
ip = va_arg(ap, (int *));
s = va_arg(ap, short);
```

The use of these macros makes the code more portable, although it does assume a certain standard method of passing arguments on the stack. In particular no holes must be left by the compiler, and types smaller than int (e.g., char, and short on long word machines) must be declared as int.

2.2.7 Side Effects, Evaluation Order

The C language makes few guarantees about the order of evaluation of operands in an expression, or arguments to a function call. Thus

```
func(i++, i++);
```

is extremely nonportable, and even

```
func(i++);
```

is unwise if func() is ever likely to be replaced by a macro, since the macro may use i more than once. There are certain XENIX macros commonly used in user programs; these are all guaranteed to only use their argument once, and so can safely be called with a side-effect argument. The commonest examples are getc(), putc(), getchar(), and putchar().

Operands to the following operators are guaranteed to be evaluated left to right:

```
,       &&      ||      ? :
```

Note that the comma operator here is a separator for two C statements. A list of items separated by commas in a declaration list are not guaranteed to be processed left to right. Thus the declaration:

```
register int a, b, c, d;
```

on a PDP-11 where only three register variables may be declared could make any three of the four variables register type, depending on the compiler. The correct declaration is to decide the order of importance of the variables being register type, and then use separate declaration statements, since the order of processing of individual declaration statements is guaranteed to be sequential:

```
register int a;
register int b;
register int c;
register int d;
```

For the same reason declaration initializations of the following type are unwise:

```
int  a = 0, b = a;
```

2.2.8 New Language Features

There are several "new" features to the C language which are not supported by all compilers, and are therefore not guaranteed to be portable. These are detailed in [5].

For example, some compilers support structure assignment statements of the following forms:

```
struct s_tag s1, s2, *sp1, *sp2;

        .......
        s2   =  s1;
        *sp2 =  *sp1;
```

A related extension to C allows the passing of structures as arguments to functions, and the returning of structures as function return values.

Some recent compilers support the void data type. This means 'no type' and is used for type casting function calls when the return value is not required. It is useful for reducing unnecessary comments from lint. For example on some systems the library routine printf is declared as being type int, yet few programs wish to examine the return value. This will attract a comment from lint unless the return value is made type void, for example:

```
(void)printf("hello world\n");
```

On those XENIX systems which do not have the void type, code containing it can be compiled by placing the following statement in a header file:

```
typedef int void;
```

This does not prevent comments from lint about ignored return values, it just enables code using void to be compiled.

2.3 Program Environment Differences

Most nontrivial programs make system calls and use library routines for various services. The sections below indicate some of those routines that are not always portable, and those that particularly aid portability.

We are concerned here primarily with portability under the XENIX operating system. Many of the XENIX system calls are specific to that particular operating system environment and are not present on all other operating system

implementations of C. Examples of this are getpwent() for accessing entries in the XENIX password file, and getenv() which is specific to the XENIX concept of a process's environment.

Any program containing hard-coded pathnames to files or directories, or user id's, login names, terminal lines or other system dependent parameters is nonportable. These types of constant should be in header files, passed as command line arguments, obtained from the environment, or by using the XENIX default parameter library routines defopen(), and defread().

Within XENIX, most system calls and library routines are portable across different implementations and XENIX releases. However a few routines have changed in their user interface:

2.3.1 System Calls

ioctl — This system call is used to alter the characteristics of a terminal line (baud rate, erase/kill processing, etc.). The routine was enhanced in XENIX Release 3, and code from previous XENIX systems will require some alteration.

open — In XENIX Release 3, this system call has been enhanced to handle FIFO's, and other functional improvements, including setting a file up for subsequent nonblocking reads.

2.3.2 Libraries

The various XENIX library routines are generally portable among XENIX systems. However there are some points to watch for.

The members of the printf family, printf, fprintf, sprintf, sscanf, and scanf have changed in several small ways during the evolution of XENIX, and some features are not completely portable. The return values from these routines cannot be relied upon to have the same meaning on all systems. Certain of the format conversion characters have changed their meanings, in particular relating to upper/lower case in the output of hexadecimal numbers, and the specification of long integers on 16-bit word machines. The manual page for printf(3S) contains the correct specification for these routines.

3. Portability of Data

Data files are almost always nonportable across different machine CPU architectures. As mentioned above, structures, unions, and arrays have varying internal layout and padding requirements on different machines. In addition, byte ordering within words and actual word length may differ.

One way to acheive data file portability is to write and read data files as one dimensional character arrays. This avoids alignment and padding problems if the data is written and read as characters, and interpreted that way. Thus ASCII text files can usually be moved between different machine types with little problem.

4. Lint

For a complete description of lint(1) see the user's guide in the XENIX Fundamentals Manual.

Lint is a C program checker that attempts to detect features of a collection of C source files which are nonportable or even incorrect C. One particular advantage over any compiler checking is that lint checks function declaration and usage across source files. Neither compiler nor loader do this.

Lint will generate warning messages about nonportable pointer arithmetic and dubious assignments and type conversions. Passage unscathed through lint is not a guarantee that a program is completely portable.

Appendix: Byte Ordering Summary

The following conventions are used below. 'a0' is the lowest physically addressed byte of the data item. 'a1' has a byte address a0 + 1. 'b0' is the least significant byte of the data item, 'b1' being the next least significant.

Note that any program which actually makes use of the following information is guaranteed to be nonportable!

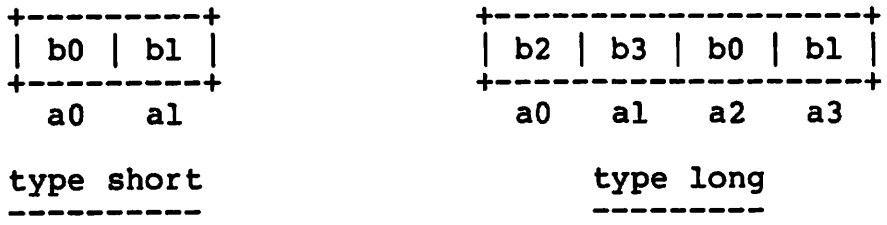

Figure 1. PDP-11 Byte Ordering

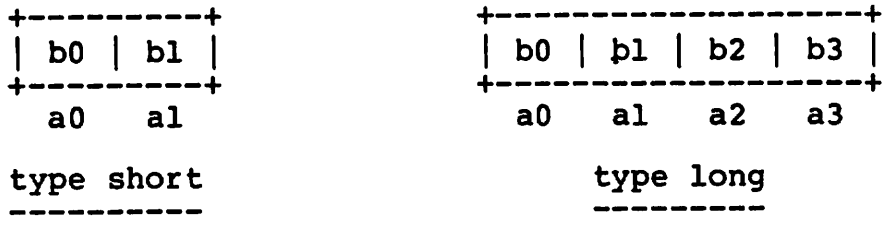

Figure 2. VAX-11 Byte Ordering

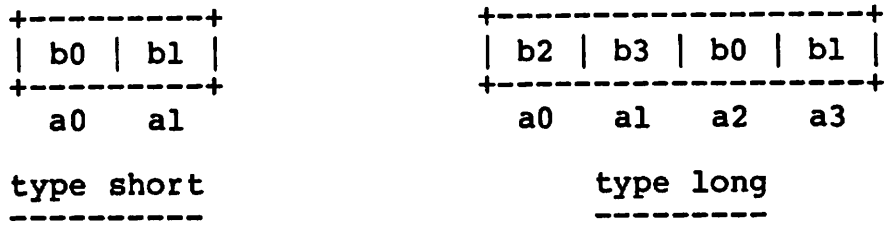

Figure 3. 8086 Byte Ordering

Appendix 1

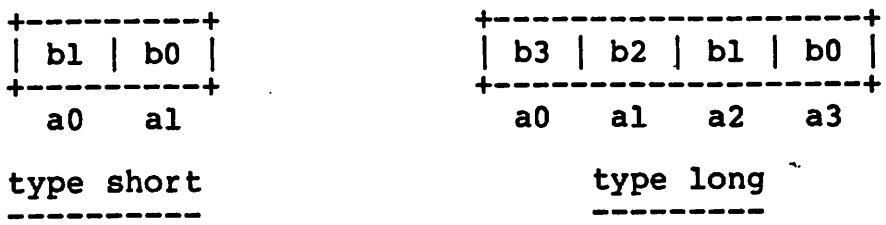

Figure 4. M68000 Byte Ordering

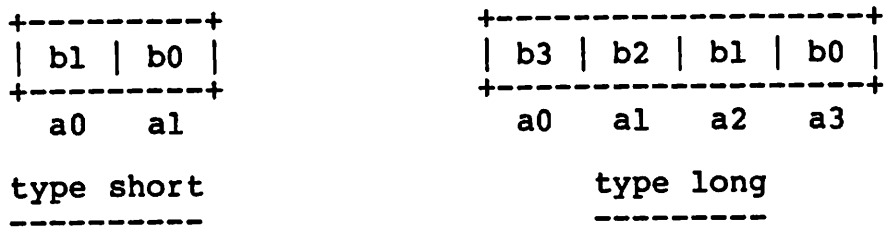

Figure 5. Z8000 Byte Ordering

Appendix 1

References

1. The C Programming Language,
   Kernighan, Brian W. and Dennis M. Ritchie. Englewood Cliffs: Prentice-Hall, 1978.

2. "A Tour through the UNIX C Compiler",
   Dennis M. Ritchie, Bell Laboratories.

3. "A Tour Through the Portable C Compiler",
   S. C. Johnson, Bell Laboratories.

4. "Sign Extension and Portability in C",
   Hans Spiller, Microsoft 1982.

5. "Recent Changes to C",
   B. R. Rowland, Bell Laboratories 1979.